UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Facial Keypoint Detection

REVIEW

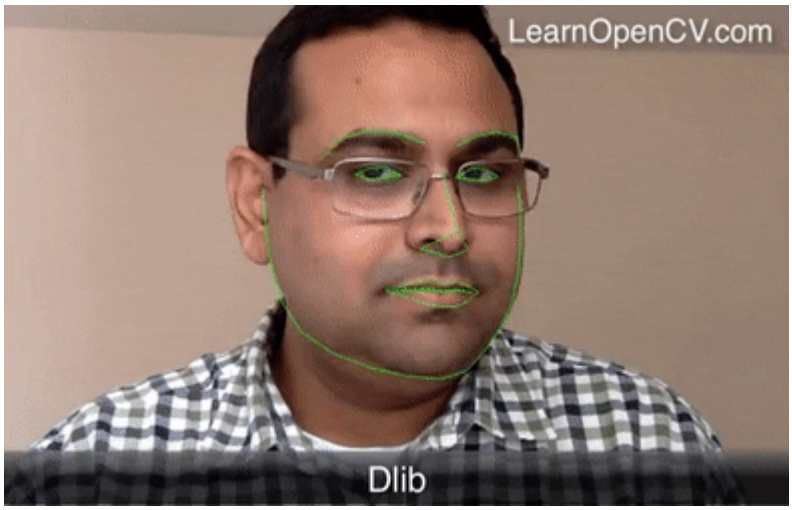
CODE REVIEW

HISTORY

Meets Specifications

You've done a fantastic job completing the Facial Keypoints Detection project. The final predictions look borderline acceptable. I appreciate your effort in experimenting and making informed decisions with the help of online resources and research papers. This is a very important trait since you'd have to do the same while working on real-world or new projects. 😊

Facial Keypoints Detection is a well-known machine learning challenge. If you want to improve this very model to allow it to work well in extreme conditions like bad lighting, bad head orientation (add appropriate PyTorch transformations like `ColorJitter`, `HorizontalFlip`, `Rotation` and more - see this post for more details), etc., the best thing you can do is to simply follow NaimishNet implementation details with some tweaks (optimizer, learning rate, batch size, etc) as per the latest improvements and your machine requirements. But for production-level performance, you can always use pre-trained models for better performance, say Dlib library provides real-time facial landmarks seamlessly. You can find a tutorial here. Here is an example:



Note: Predicted keypoints are joined with lines here.

Here are some advanced research works involving facial landmarks:

- Style Aggregated Network for Facial Landmark Detection (Code)
- Supervision-by-Registration: An Unsupervised Approach to Improve the Precision of Facial Landmark Detectors (Code)
- Teacher Supervises Students How to Learn From Partially Labeled Images for Facial Landmark Detection (Code)
- A Fast Keypoint Based Hybrid Method for Copy Move Forgery Detection
- Disguised Face Identification (DFI) with Facial KeyPoints using Spatial Fusion Convolutional Network
- Berkeley team's attempt at beating the Facial Keypoints Detection Kaggle competition
- Facial Keypoints Detection using the Inception model

Facial keypoint prediction pipeline can be extended to human poses, hand poses and more to help intelliget systems like robots, automatic anomaly detectors understand the orientation of subjects in CCTV footage, etc. Here are some works based on keypoint prediction:

- OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields (Code)
- PoseFix: Model-agnostic General Human Pose Refinement Network (Code)
- SRN: Stacked Regression Network for Real-time 3D Hand Pose Estimation (Code)
- Hand Pose Estimation: A Survey

Keep up the good work and good luck with the rest of the nanodegree! 😊👍

Files Submitted

The submission includes models.py and the following Jupyter notebooks, where all questions have been answered and training and visualization cells have been executed:
2. Define the Network Architecture.ipynb, and
3. Facial Keypoint Detection, Complete Pipeline.ipynb.

Other files may be included, but are not necessary for grading purposes. Note that all your files will be zipped and uploaded should you submit via the provided workspace.

`models.py`

Define a convolutional neural network with at least one convolutional layer, i.e. `self.conv1 = nn.Conv2d(1, 32, 5)`. The network should take in a grayscale, square image.

Notebook 2: Define the Network Architecture

Define a `data_transform` and apply it whenever you instantiate a DataLoader. The composed transform should include: rescaling/cropping, normalization, and turning input images into torch Tensors. The transform should turn any input image into a normalized, square, grayscale image and then a Tensor for your model to take it as input.

Depending on the complexity of the network you define, and other hyperparameters the model can take some time to train. We encourage you to start with a simple network with only 2 layers. You'll be graded based on the implementation of your models rather than accuracy.

Select a loss function and optimizer for training the model. The loss and optimization functions should be appropriate for keypoint detection, which is a regression problem.

Train your CNN after defining its loss and optimization functions. You are encouraged, but not required, to visualize the loss over time/epochs by printing it out occasionally and/or plotting the loss over time. Save your best trained model.

After training, all 3 questions about model architecture, choice of loss function, and choice of batch_size and epoch parameters are answered.

Your CNN "learns" (updates the weights in its convolutional layers) to recognize features and this criteria requires that you extract at least one convolutional filter from your trained model, apply it to an image, and see what effect this filter has on an image.

After visualizing a feature map, answer: what do you think it detects? This answer should be informed by how a filtered image (from the criteria above) looks.

Notebook 3: Facial Keypoint Detection

Use a Haar cascade face detector to detect faces in a given image.

You should transform any face into a normalized, square, grayscale image and then a Tensor for your model to take in as input (similar to what the `data_transform` did in Notebook 2).

Well done transforming the face image into a normalized, grayscale image and passing it through the model as a Tensor!

After face detection with a Haar cascade and face pre-processing, apply your trained model to each detected face, and display the predicted keypoints for each face in the image.

The model has been applied and the predicted key-points are being displayed on each face in the image. Your model predictions look borderline acceptable. Very well done! 😊

I appreciate your idea to add padding to the faces. It is a very effective trick. Looking at the training images in Notebook 2, I bet you would agree with me that the faces in the dataset are not as zoomed in as the ones Haar Cascade detects. This is why you MUST grab more area around the detected faces to make sure the entire head (the curvature) is present in the input image. You can do the padding in a generic way, without having to use a constant padding value, with the following code:

```
margin = int(w*0.3)
roi = image_copy[y-margin:y+h+margin, x-margin:x+w+margin]
```

or

```
margin = int(w*0.3)
roi = image_copy[max(y-margin,0):min(y+h+margin,image.shape[0]),
                 max(x-margin,0):min(x+w+margin,image.shape[1])]
```

```
max(x-margin,0):min(x+w+margin,image.shape[1])]
```

Please do not get discouraged by the quality of your model's predictions. Currently, you can try tuning the scalars (standard deviation and mean) in the de-normalization step manually to suit your image:

```
predicted_key_pts = predicted_key_pts*50.0+100
```

Be aware, a "good" model doesn't enforce us to manually tune these values.

Although NOT a rubric of the project, you can take up the task of mapping the points on to the original image (instead of plotting the points on separate faces) after you are done with this project. It is a simple yet mind-tingling programming exercise, give it a go. 😉

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START